An Introduction to Microservices

The essential concepts that every developer should know

Hello and welcome to the inaugural blog of the Microservice Geeks publication. This is a place to share your experience and learn as a community. And what better way to get started than to dive right into this fascinating thing called “microservices”.

A Brief History

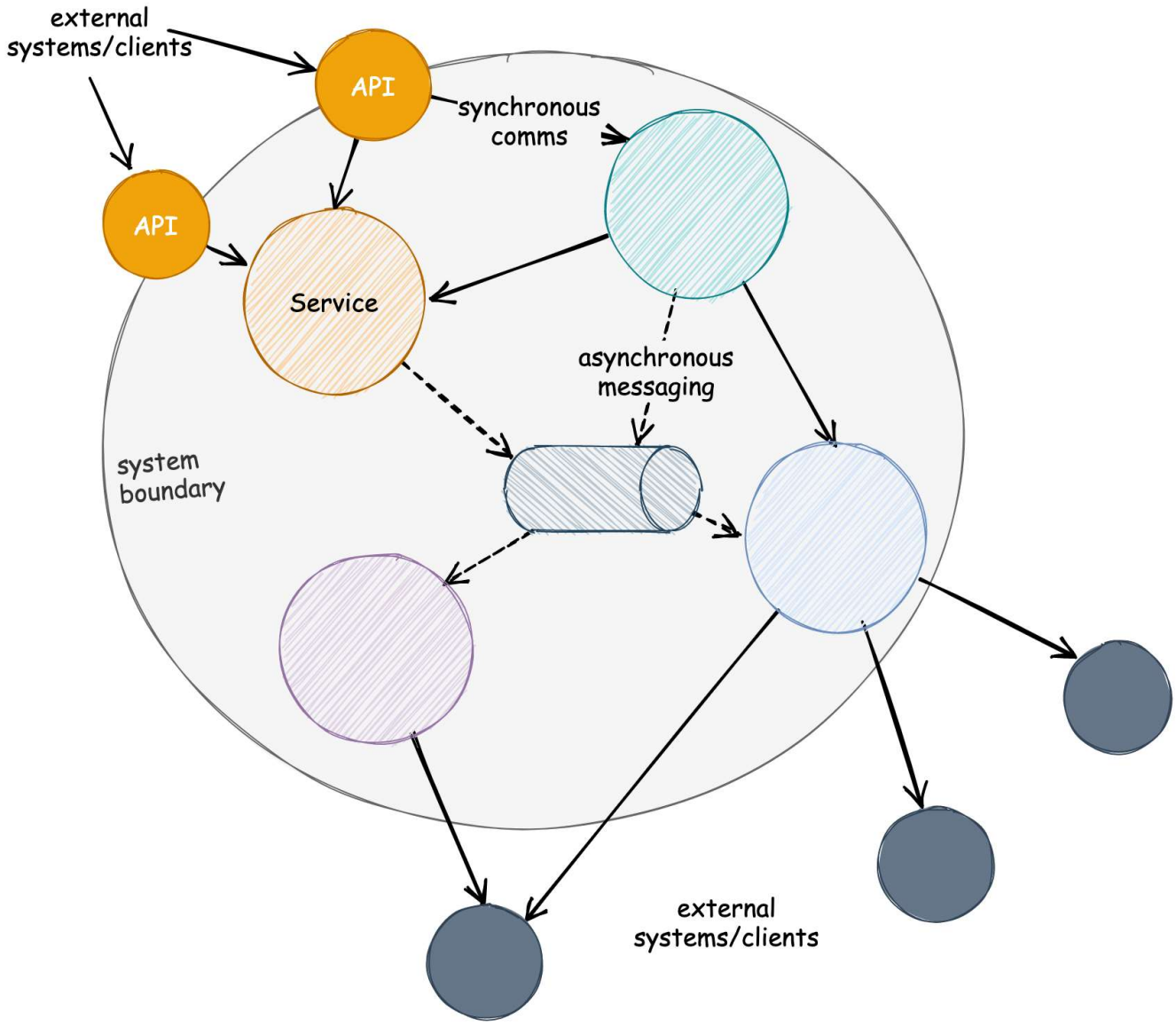
The term **microservices** was first coined by Dr. Peter Rodgers in 2005 and was initially known as “micro web services”. The main driver behind “micro web services” at the time was to break up single large “monolithic” designs into multiple independent components/processes, thereby making the codebase more granular and manageable.

Modular, distributed applications date for back several decades. And in this regard, microservices are not a new concept. However, what popularised microservices was the principles governing how they were designed and the way they were consumed. While conventional distributed systems of that era relied on proprietary communications protocols, microservices took advantage of open standards such as HTTP, REST, XML and JSON.

A Simple Definition

A **microservice** is a small, loosely coupled, distributed service. It is part of a broader **microservices architecture**, comprising a set of loosely coupled microservices that operate together to solve a common goal. A collection of microservices can be regarded as a **system**.

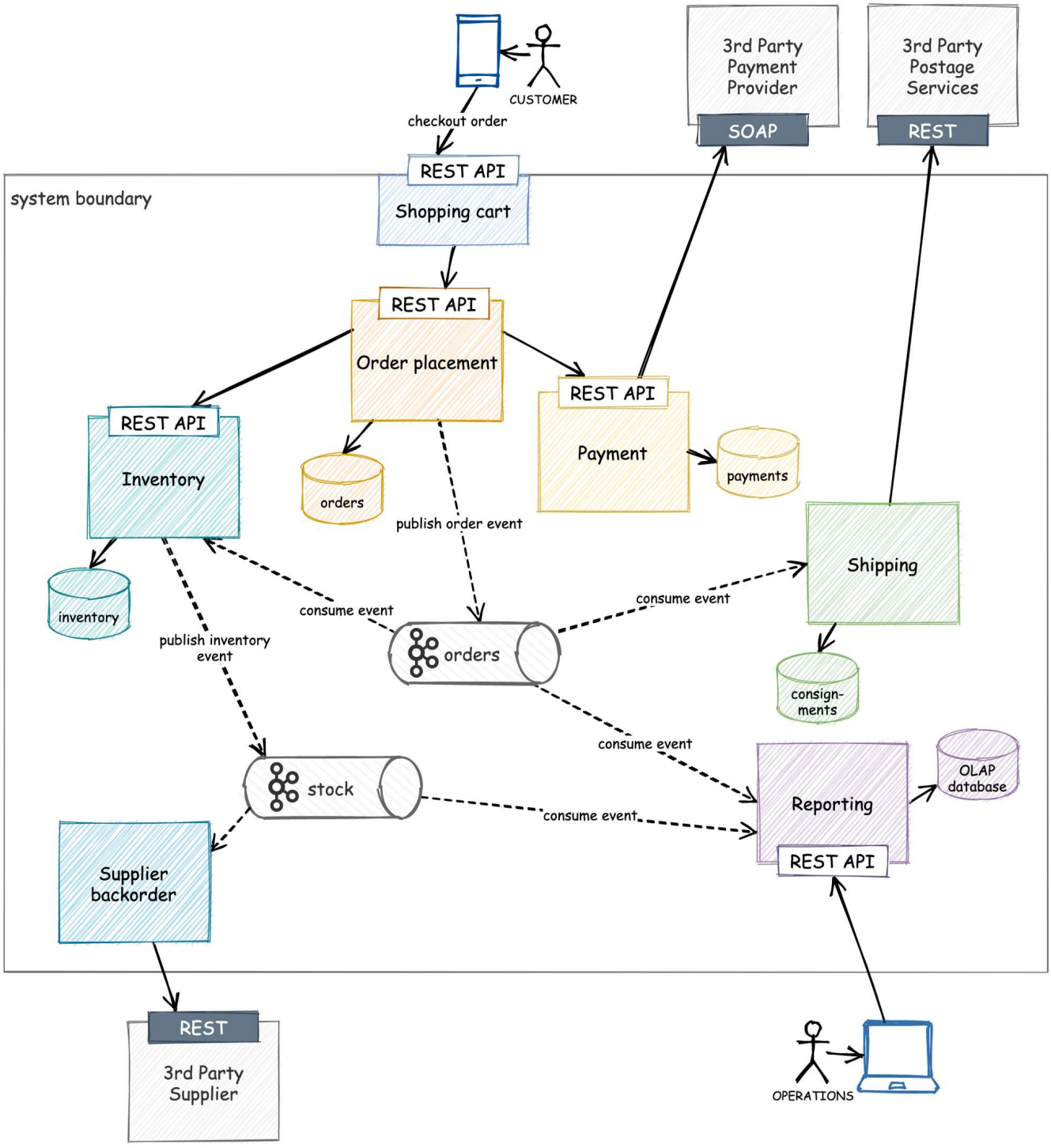
The diagram below is a super-simple reference model for a microservices architecture. It illustrates a hypothetical system comprised of several granular services that communicate either **synchronously** — via internal API calls, or **asynchronously** — via message passing with a help of a message broker. The entire deployment is contained within a notional **system boundary**. External systems (users, applications, B2B partners, etc.) can interact with the microservices only through a set of externally-facing APIs — commonly referred to as an **API gateway**. Services within the boundary can freely consume external services as necessary.



Microservices references model

Microservices evolved as a solution to the scalability challenges with monolithic architectures. A microservice application is typically small: in the order of thousands of lines of code. By comparison, a "monolith" is typically an application comprising hundreds of thousands of lines of code.

This is only a rough comparison, of course — derived from empirical observations. There is nothing implying that a microservice must be tiny, or that a monolith is huge. What generally sets them apart is the number of responsibilities that developers *tend* to cram into them. **A microservice will typically handle a small handful of related responsibilities**; for example, dealing with order processing. Another microservice might be responsible for shipping. Another might take care of payment. Collectively, these microservices might function as a complete e-commerce system. An example of this is illustrated below.



An example of a microservices architecture

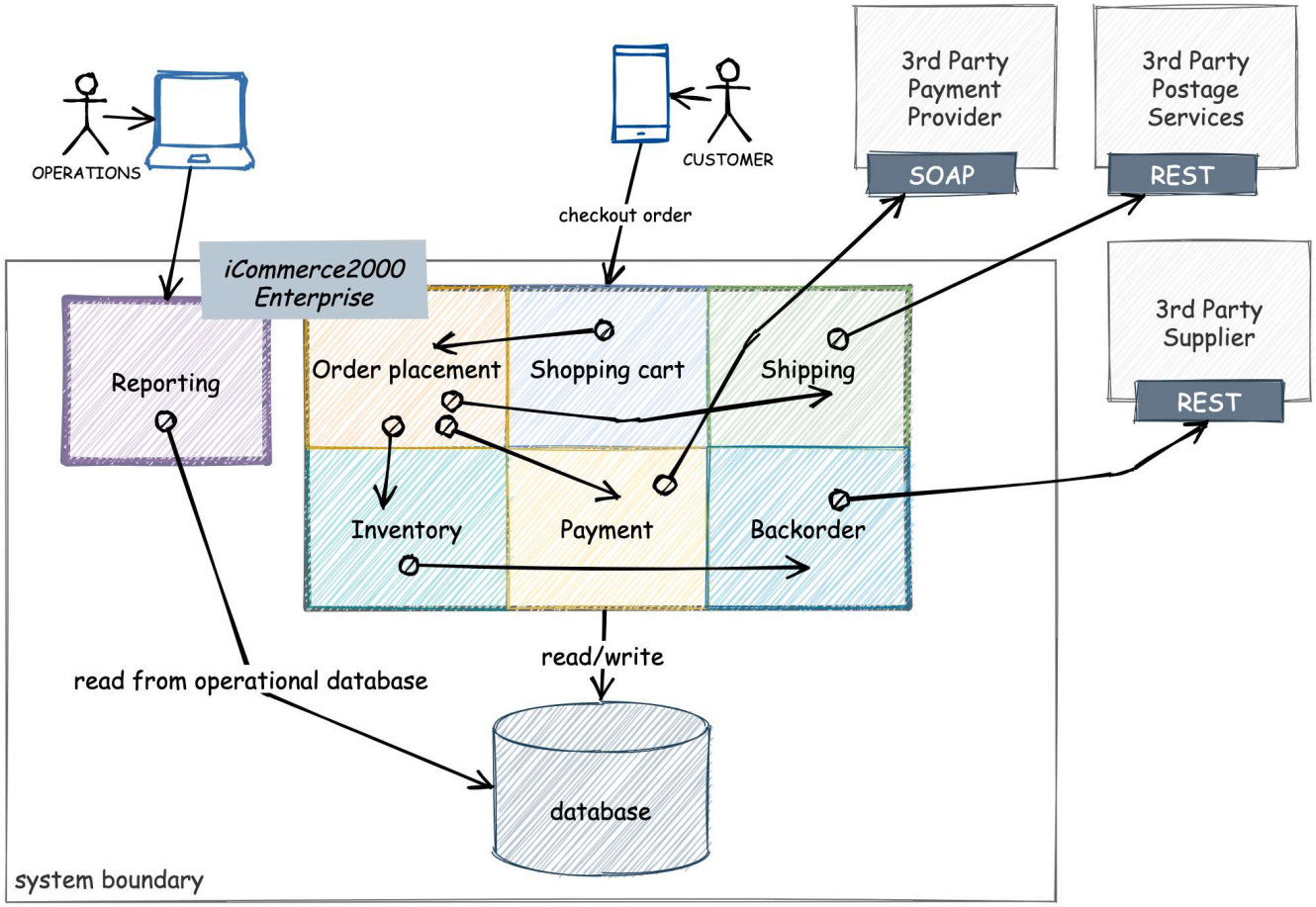
This is a breakdown of a simple but functional e-commerce platform. Each microservice in the picture serves a specific role, taking complete care of an isolated domain — such as *shopping cart*, *inventory*, *order placement*, *payments*, *shipping*, *reporting* and *supplier backorder*. The *shopping cart* service acts as a lightweight **BFF** (back-end for front-ends), acting as an API gateway of sorts to insulate the client application from the business logic necessary to fulfil orders.

Microservices communicate synchronously only when absolutely necessary; for example, the *order placement* service will check with *inventory* and *payments* before asynchronously registering the order by publishing an event on a Kafka topic. We want the inventory and payment check to be synchronous, as the failure of either check should prevent the order from progressing. Once an order is registered, the rest of the fulfilment can be entirely asynchronous. It doesn't really matter when the order is shipped or if the backorder is triggered behind the scenes — none of these actions impact the customer's checkout journey. Consequently, if the *supplier backorder* service is temporarily down or if the *shipping* service is backlogged for whatever reason, the rest of the system can operate as normal, albeit backorders will not be placed with our suppliers until the outage is fixed.

***Note**: On the point of synchronous vs asynchronous communication: **asynchronous is preferred** as it reduces coupling, but **synchronous communication is sometimes necessary**. It would be poor customer experience if we allowed the order to continue without so much as checking whether we had sufficient stock.*
*But **synchronous communication does not guarantee consistency**: it may be possible for an inventory check to pass, only to fail later when the order is fulfilled in the background. (Concurrent orders might individually verify the presence of stock, which may not be sufficient to cover all orders at the point of fulfilment.)*

The solution is to use the Saga Pattern, although this is generally considered an advanced integration pattern — one that we will ignore for the time being.

By contrast, a monolith will typically take care of all these concerns in a single application. The diagram below illustrates the application of a fictitious (but highly plausible) **iCommerce 2000 Enterprise** e-commerce platform.



An example of a monolithic application

In the model above, all components with the exception of the reporting system are packed into a single process. Reporting almost always sits outside the main monolith because it is often accomplished using an off-the-shelf software package. Of course, that doesn't stop it from accessing the main operational database of our monolith — an architectural crime committed by most enterprise architects.

We kept the rest of the components the same. In Java, these might be packages or modules. Unfortunately, more often than not, components in a monolith are nothing more than a handful of related classes with very loose laws governing their communication. A class may reach out to any other class; the only thing maintaining encapsulation is the distinction between public and private fields and methods.

Because modules communicate synchronously within the same process, the failure of one module implies the failure of the entire system. It is also easy to see how adding features to this application impacts the overall complexity — the number of lines grows exponentially.

Reasons for Building Microservices

To understand why we would venture down this path, consider the typical challenges inherent in monolithic applications:

1. For every change, the entire application needs to be rebuilt and redeployed, irrespective of how large or small the change is. 15–30 minute build times are not uncommon for large applications.
2. A small change in one part of an application has the potential to break the entire system.
3. As the application grows in size, its parts tend to become more intertwined and the codebase becomes difficult to understand and maintain.
4. Large applications have a correspondingly large resource footprint — they typically consume more memory and require more computing power. As a result, they must be hosted on large servers with sufficient resource capacity. This also limits their ability to scale.
5. They also tend to have a slow startup time, which is not ideal, given that even the tiniest changes tend to require a complete redeployment. They are less suited for the Cloud and cannot easily take advantage of ephemeral computing, such as spot instances.
6. A single technology is used to implement the entire application, often a compromise between generality and the needs of specific areas of the application. Java and .Net are likely candidates for monoliths because they are among the best “all-rounder” languages, not because they are amazingly good at any particular task.
7. Team scalability is naturally constrained by a large codebase. The more complex the application (in terms of internal dependencies), the more difficult it is to comfortably accommodate large teams of developers, without people stepping on each other’s toes.

The main drivers behind microservices adoption

Benefits of Microservices

1. **Scalability.** Individual processes in a microservices architecture can scale to meet their demands. Smaller applications can scale both **horizontally** (by adding more instances) as well as **vertically** (by increasing the resources available to each instance).

2. **Modularity.** An obvious advantage of having smaller, standalone applications is that the **physical separation between processes forces you to address coupling at the forefront of your design**. Each application becomes responsible for fulfilling fewer responsibilities, resulting in a more compact, more cohesive code. It may be argued that monoliths can (and should) also be designed in a modular fashion, with coupling and cohesion in mind; however, the in-process nature of monoliths means that developers are free to short-circuit "soft" boundaries within the application and break encapsulation, often without realising it.

3. **Tech diversity.** Microservices applications are autonomous units that communicate over open standards. This means that the **technology choices behind microservices implementations are far less significant compared to a monolith**; which is to say, the choices matter for the microservice in question, but do not concern the rest of the system. It is not uncommon to see a single microservices architecture implemented with a blend of technologies — Java and Go for business logic, Node.js for API gateways and presentation concerns, and Python for reporting and analytics.

4. **Opportunities for experimentation.** An extension of the previous point, a microservice is autonomous and may be designed and developed separately from its friends. It will have its own database, separate from the others'. It can be built in a language that is best suited for its purpose. This autonomy lets the team safely experiment with new technologies, approaches and processes, and **fail-fast** — should one of the experiments fail, its mitigation costs are relatively low — confined to a single service.

5. **Eases migration.** We've all worked on large monolithic software systems that were built on two-decade-old technology, being too difficult and risky to update. I personally recall working on a project in 2018 where the team was stuck on Java 6 — held back by complex library dependencies, lack of proper unit tests and the resulting risk of migration. This is rarely the case with microservices. Smaller codebases are easier to refactor and even poorly written individual microservices don't hold up the rest of the system.

6. **Resilience and availability.** When a monolith goes down, the business stops. Of course, the same might be said for poorly designed, strongly coupled microservices with complex interdependencies. However, good microservices architecture emphases loose coupling, where services are autonomous, fully own their dependencies, and minimise synchronous (blocking) communication. When a microservice goes down, it will invariably impact some part of the system and affect certain users, but it will typically allow other parts of the system to operate.

Limitations of Microservices

1. **Atomic versioning.** Versioning a monolith tends to be straightforward, as the codebase lives in a single repository, along with all related tags and branches. When you check out a version, you can be fairly sure that all components are compatible and can be safely deployed together. Microservices tend to be developed independently and housed in separate repos. But they still have to communicate. It is a lot more **difficult to keep track of versions and ensure compatibility when services are not co-versioned**. The same challenges that apply to code, also apply to configuration.

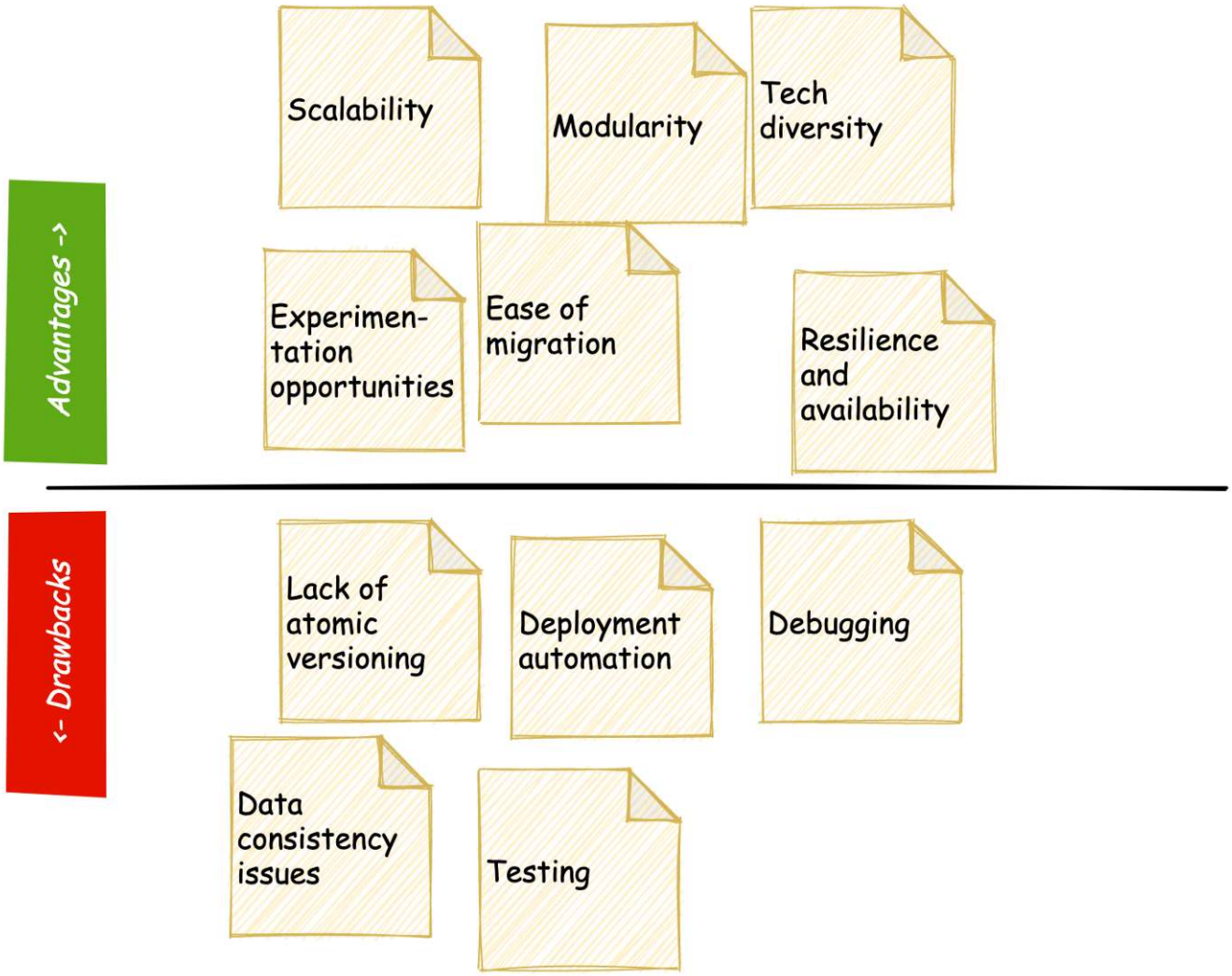
2. **Deployment automation.** You might get away with manually copying WAR or EAR files over to an Application Server in a data centre—when you are deploying one application to a pair of servers once a month. This process spells a disaster for microservice architectures. When your team operates a fleet of several dozen

microservices that routinely undergo change, manual processes will not cut it. **Microservices need a mature DevOps philosophy and CI/CD processes and infrastructure.**

3. **Debugging.** It is a lot easier to debug the interactions between components of a system when they communicate in the same process, especially when one component simply calls a method on another. This is usually just a matter of attaching a debugger to the process, stepping through the method calls and watching variables. There is no straightforward equivalent of this in microservices. Service communications are notoriously difficult to trace and piece together, requiring additional tooling, infrastructure and complexity. You don't want to find yourself in the middle of a system-wide outage in a microservices architecture.

4. **Data consistency.** A monolith operating over a single relational database has the benefit of ACID. In other words, **transactions**. Transactions take on a very different form in distributed systems. Generally speaking, consistency is a lot more difficult to achieve, especially given the types of failures that occur in distributed systems.

5. **Testing.** A monolith can be readily tested as a complete system, be it manually or (preferably) with an automated test suite. Testing a single microservice does not paint the whole picture: even if a service is functioning to its specification, it does not imply that the entire system will work as designed. The only way to be sure is to run more complex integration and end-to-end (or acceptance) tests.



The advantages and drawbacks of a microservices architecture

Conclusion

In summary, the term “microservices” stands for an alternative architectural paradigm that composes complex systems from small, granular services. By breaking the problem down to more bite-sized chunks, **a microservices architecture simplifies software maintenance, allows for improved scalability and ultimately leads to more robust and sustainable solutions**. Microservices are also not without their challenges: it is a lot more difficult to ensure data consistency in a distributed system and debugging misbehaving services in production is both a science and an art. Still, the advantages cannot be ignored, and if your organisation hasn’t yet started its transition to microservices, chances are that it soon will. Best be prepared.